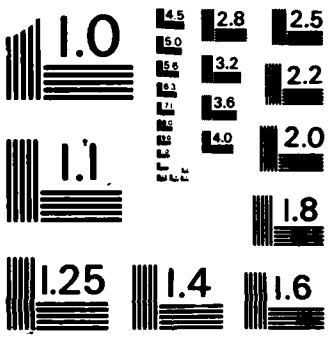
AD-A171 828 PROCEDURES FOR CAUSE ORIENTED ANALYSIS AND CONSEQUENCE ORIENTED ANALYSIS(U) MITRE CORP MCLEAN VA A L MCFARLAND ET AL. JUN 86 DOT/FAA/PM-86/21 1/1

UNCLASSIFIED DTFA01-84-C-0001 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

DOT/FAA/PM-86/21

Program Engineering
and Maintenance Service
Washington, D.C. 20591

Procedures for Cause Oriented Analysis and Consequence Oriented Analysis

(2)

Dr. Alvin L. McFarland
Dr. Yan-Shek Hoh
Dr. Gerald A. Dorfman

The MITRE Corporation
McLean, Virginia 22102

June 1986
Final Report

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof.

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/PM-86/21		

4. Title and Subtitle	5. Report Date
PROCEDURES FOR CAUSE ORIENTED ANALYSIS AND CONSEQUENCE ORIENTED ANALYSIS	June 1986
	6. Performing Organization Code
	8. Performing Organization Report No.
7. Author(s) Dr. Alvin L. McFarland; Dr. Yan-Shek Hoh Dr. Gerald A. Dorfman	
9. Performing Organization Name and Address The MITRE Corporation 1820 Dolley Madison Boulevard McLean, VA 22102	10. Work Unit No. (TRAIS) W-46
	11. Contract or Grant No. DTFA01-84-C-0001
12. Sponsoring Agency Name and Address U. S. Department of Transportation Federal Aviation Administration 800 Independence Avenue, S.W. Washington, D. C. 20591	13. Type of Report and Period Covered Final Report
	14. Sponsoring Agency Code APM-410

15. Supplementary Notes

16. Abstract

Cause and consequence oriented analyses are techniques for safety analysis of software. They are particularly important for analyzing software whose inadequate performance could result in hazard to life or property. This report presents procedures for performing these analyses.

17. Key Words	18. Distribution Statement
software analysis; software testing; system safety analysis; safety critical software; mission critical software; cause oriented analysis; consequence oriented analysis	This document is available to the public through the National Technical Information Service, Springfield, Virginia 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	26	

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

1. INTRODUCTION.

Increasingly modern systems include software that performs safety critical functions[1,2]. At present, there is a lack of concurrence on a specific set of techniques that can be readily applied to ensure the development of safe software. Traditional system safety* techniques for hardware are inadequate for software safety analysis. Attempts have been made, with some success, to adapt hardware safety analysis techniques to software[5]. Reference 2 presents brief discussions of four software safety analysis techniques:

- software fault tree (soft tree) analysis
- software sneak circuit analysis
- nuclear safety cross-check analysis (NSCCA)
- safety analysis using Petri nets

These techniques are in various stages of development and differ in their utility for analyzing various aspects of software safety. Petri nets, for example, are useful for analyzing concurrency problems in parallel processing systems[6] but their theory is still under development[7].

This paper presents procedures for a variant of soft tree analysis[1,5] called cause oriented and consequence oriented analyses.

2. DISCUSSION.

Cause and consequence oriented analyses are system safety techniques for safety analysis of software. They are performed as follows:

A list of failure vulnerabilities is prepared. This list contains initiating events and undesired end results. The list may be developed from: items of special concern to members of the technical community; items solicited from designers and other testers of the logic; and items gleaned from software trouble reports, in-depth analyses of other items, and other sources. The list is then partitioned into two sublists -- one listing initiating events, the other listing undesired end results.

---

* System safety analysis is concerned with assuring that a system does not cause hazards while seeking to achieve its mission[3]. System safety analysis also seeks to assure that a system will prevent erroneous inputs and unanticipated events from causing hazards. A hazard is a condition that can lead to an unplanned event or series of events that result in death, injury, illness, or damage to or loss of property[4]. General system safety program requirements are discussed in Reference 4.

In cause oriented analysis, each item on the list of undesired end results is used to begin the development of a logic tree. The development proceeds in what might be considered a backward direction, starting with the undesired end result and searching for a basic software fault that could cause that undesired end result. In a similar way, each item on the list of initiating events is used to begin a logic tree for consequence oriented analysis. The analysis proceeds in a forward direction, starting with the initiating event and searching for a software fault that could lead to a hazardous situation.

Using the above process, trees are developed of both the cause oriented and the consequence oriented type. Then a single path leading from the top of the tree down to a single terminating branch is taken as a single situation for study. The blocks in this path fully describe that specific situation. The software logic is then traced in the proper order as called for by this particular path in the logic tree. The logic tracing is performed by an analyst using the appropriate flow charts or program design language. The analyst observes whether or not the software logic works correctly in that particular situation. If so, he or she goes on to the next path in the logic tree. If not, a software fault has been found; and it is documented in detail. Ideally, the logic tracing is performed for every situation on each tree. In practice, resource limitations may necessitate performing the logic tracing on only a subset of the situations.

In employing cause and consequence oriented analyses, as with any software safety technique, checklists of safety criteria should be employed. Several useful safety checklists are contained in References 2 and 8.

A specific application of cause oriented and consequence oriented analyses and more description of how to conduct these kinds of analyses are presented in Appendix A.

3. REFERENCES.

1. N. G. Leveson and P.R. Harvey (1983), "Software Fault Tree Analysis," The Journal of Systems and Software, 3, 173-181.

2. Air Force Inspection and Safety Center (1985), "Software System Safety," AFISC SSH 1-1, Norton Air Force Base, CA: Department of the Air Force.

3. N. G. Leveson and P.R. Harvey (1983), "Analyzing Software Safety," IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, 569-579.

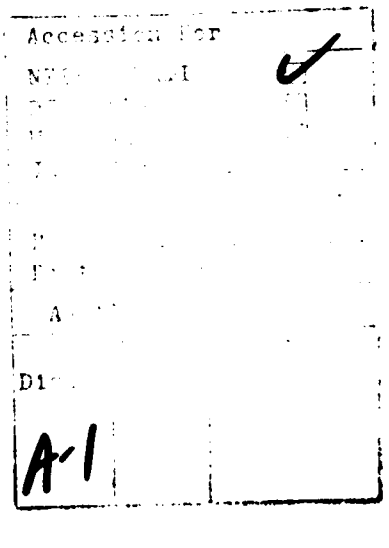
4. U.S. Department of Defense (1984), "System Safety Program Requirements," MIL-STD-882B, U.S. Government Printing Office.

5. J. W. McIntee (1983), "SOFT TREE Fault Tree Technique as Applied to Software," Eglin Air Force Base, FL: AD/SES.

6. J. L. Peterson (1978), "An Introduction to Petri Nets," Proceedings of the National Electronics Conference, Vol. 32, 144-148, Oak Brook, IL: National Engineering Consortium, Inc.

7. International Workshop on Timed Petri Nets, Torino, Italy, July 1-3, 1985 (1985), Silver Spring, MD: IEEE Computer Society Press.

8. M. H. Bell (1984), "Software Safety Analysis," in Proceedings COMPCON Fall '84, The Small Computer (R)Evolution, Arlington, VA, 16-20 September 1984, Silver Spring, MD: IEEE Computer Society Press.



APPENDIX A
AN APPLICATION AND DETAILED DESCRIPTION OF CAUSE AND CONSEQUENCE ORIENTED ANALYSES

Background

Computer controlled systems containing highly complex software are being found more and more frequently in safety critical applications. Such systems include aircraft flight control systems (including systems capable of automatically landing large passenger aircraft), manned spacecraft guidance systems, nuclear power plant monitoring and control systems, air traffic control automation systems and rapid rail vehicle control systems. These systems have the following common characteristics:

- The software controls a physical system in real-time.
- The overall system is highly complex.
- There are many interactions between individual subsystems.
- Software errors have the potential for directly causing life-threatening situations.

Verifying the safety of software in these systems is a significant problem for implementors of these systems.

The discipline of system safety has become well developed. Its techniques have been used in diverse fields to analyze the safety of large systems. But these techniques have traditionally been used to study physical failures of physical systems. Only within the past few years have researchers and analysts discussed use of the established system safety techniques for safety analysis of software. To date, there seems to have been little effort to apply these techniques to the software area. This paper reports on a safety study conducted on a complex software design. The study used adaptations of some of the traditional system safety techniques. The paper summarizes the lessons learned from this study and comments on the applicability of some of the techniques for safety analysis of other safety critical software systems.

The work described in this Appendix was performed by Dr. A. L. McFarland and Dr. Y-S Hoh. It was presented at the Sixth International System Safety Conference held 26-30 September 1983 in Houston, Texas. This activity was supported by the Office of Systems Engineering Management of the Federal Aviation Administration under Contract DTFA01-82-C-10003. Views represent those of the authors and are not necessarily official policy of the U. S. Government.

Traditional Approaches to Software Verification

In the past, when complex software was used in safety critical applications, the software was generally verified by doing test, test, test and more test. Such test and verification activities included the following:

- Desk checks using a checklist.
- Design walk-throughs in which a group of designers review software modules one at a time.
- Unit tests.
- Simulation (possibly using Monte Carlo techniques to expose the software to random combinations of conditions).
- Bench tests with an environment test set to provide simulated inputs to the software when running in its host computer.
- Live or flight tests.

Some very complex software systems verified using these approaches have demonstrated admirable safety records. Nevertheless, these approaches have several limitations.

Software programs of the type mentioned above are so complex, and involve so many variables and conditions, that it is impossible to exhaustively test every combination of inputs. To illustrate, suppose that a real-time software system is a cyclic process which performs the same basic tasks cycle after cycle. To be exhaustive, it is not enough to test the system with all combinations of inputs which it could experience on one cycle. To uncover all possible faults, one would also have to consider all possible sequences of one-cycle combinations of inputs. Even when using a computer to generate test scenarios, it is generally not possible to be exhaustive because the number of combinations of inputs is still prohibitive. In addition, it is very difficult to build an automatic results analyzer, which can judge whether or not a fault has occurred from the results of such test runs. And the number of runs to exhaustively cover the test domain makes it impractical for a person to manually review the results.

In safety analysis of software the perspective differs somewhat from that of most software testing. Most of the testing is to ensure that the software meets the conditions of a requirements specification. But in the case of safety critical software it is implied (if not stated explicitly) that the software should handle all conditions with which it may be confronted and should not

permit or cause any dangerous conditions. The original requester will not have identified every conceivable software fault or failure mode and written specific requirements that the software must avoid each of them. The purpose of software safety verification is to identify hazardous software faults, when it is not known in advance what their nature is or even whether they exist.

Manually creating test scenarios is a very demanding task which requires the test planner to have great imagination. Traditional test strategies depend upon testing at the extremes of the ranges of input variables, testing to insure that every branch has been taken at least once, testing to ensure that each statement has been executed at least once, and the like. These approaches will reveal many software faults; but others will not be found. One actually has to imagine the nature of the error in advance, to be able to create a test scenario which will reveal the presence of some of these less-obvious faults.

Building an environment test set for a large real-time system which has many different inputs can be a very expensive and challenging task. Achieving time synchronization between the test set and the subject system can be difficult. So also can be the task of maintaining coherency between all of the inputs which must be simulated by the test set. In many cases, just the volume of inputs that must be generated makes it difficult for the test set to keep up with real time. Many of the features of a system which might be tested require special provisions, within the test set. Because there may be so many of these special provisions, and because they may be so difficult to implement, many of them may not be incorporated. The test set may include random generators for some variables, but even randomizing the variables that are simulated will not always compensate for failing to exercise some of the features of the subject system. Thus, the environment test set is unable to generate enough conditions to reveal all software faults.

Live or flight tests can be very valuable in identifying unforeseen software errors; but these suffer from the major limitations that they are very expensive and time consuming. Generally, live tests are resource constrained so that only a limited number of tests can be conducted. Many of the safety-critical situations that the system might experience may be too hazardous for live tests. Such might be the case for certain types of tests involving nuclear power plants. In other cases it may not be feasible to conduct flight tests in advance of full operation. The NASA space shuttle, which was manned for the first launch, was an example. In live tests it may be difficult to control the test scenario to achieve pre-specified conditions.

Each of the test approaches discussed above has limitations; even when all are used for verification of a particular system, they will

probably not reveal all latent faults. The major difficulty is that practically all of these methods require the test planner to imagine the software fault beforehand so that he can create a test scenario which will have a chance of revealing that error. Certainly, Monte Carlo approaches or live testing will reveal some new software errors by accident; but many will still remain hidden. The history of software testing is full of statements like the following by software testers when an error is discovered after implementation. "I never thought about building a test for that kind of condition."

What is needed is an approach to help the test planner or software verifier identify potential failure modes. This is the role that system safety techniques can play in verification of software in safety-critical applications. These techniques can help the analyst identify unforeseen faults through organized approaches. These techniques, by approaching software verification from different perspectives, have the potential for finding software faults not revealed by other methods. These techniques would enhance, but not replace, the usual methods of software verification.

An Aircraft Collision Avoidance System Coordination Logic

During the period 1978-1980, the Federal Aviation Administration (FAA) had under development two aircraft collision avoidance systems - an airborne system and a ground-based system. The airborne system was a sophisticated, self-contained system, intended for use on large aircraft. The system could determine positions of neighboring aircraft by sending out interrogations and listening to replies from the aircraft. It tracked the positions as a function of time to determine velocities. By projecting the neighboring aircraft ahead in time, the system could tell when another aircraft was a potential collision threat. When such a situation was observed, the system would display an instruction to the pilot for avoiding the other aircraft. The instruction was "Climb", "Descend", "Don't Climb", "Don't Descend" or another similar instruction. If the other aircraft also carried the airborne collision avoidance system, the two airborne systems would coordinate the resolutions so that the two pilots would receive complementary instructions (e.g., one would receive "Climb", the other "Descend").

The ground-based system worked in conjunction with a ground radar. The aircraft within coverage of the radar were tracked, potential collisions were detected, and resolution actions were calculated through algorithms existing in the ground radar's computer. When avoidance maneuvers were required, the ground sent them to the appropriate aircraft by a digital data link which was an integral part of the radar's design. These ground instructions could be received and displayed by a rather simple avionics device which

could be carried by small, privately-owned aircraft. The airborne collision avoidance system was also able to receive instructions from the ground radar.

The FAA established requirements for a coordination scheme which would determine whether the airborne system or the ground system should have responsibility in any given situation and would ensure compatibility of instructions given to all aircraft in a conflict. Preliminary and refined designs for such a coordination scheme are documented in References 1 and 2.

Figure 1 illustrates the scope of the coordination logic. There are three types of avionics of concern:

- Airborne collision avoidance system. This system can observe other aircraft carrying any of the three types of avionics and can generate instructions for its own pilot. It can also receive instructions from the ground collision avoidance system.

- New avionics that are able to receive instructions from the ground collision avoidance system and display them to its own pilot. These avionics are also able to report, either to an airborne collision avoidance system or to a ground collision avoidance system, what instructions are currently being displayed to its own pilot.

- Today's avionics. This is a simple transponder which responds to interrogations with replies which indicate own aircraft's identity and altitude. An aircraft that carries these avionics is not able to receive instructions, but is able to be observed by either the airborne or the ground collision avoidance system.

The radars supporting the ground collision avoidance system have limitations in their coverage of the airspace. There is a maximum useful range and they cannot provide coverage below the line of sight as established by the local horizon. The radars provide overlapping coverage in much of the airspace.

The coordination logic has the following capabilities:

- It will assign one and only one system (the airborne system or one of the ground systems) to be responsible for each conflict between two aircraft.

- It will handle conflicts involving any number of aircraft simultaneously.

---

1 "Report of the FAA Task Force on Aircraft Separation Assurance", Federal Aviation Administration, Report No. FAA-EM-78-19, Vols. I, II, III, January 1979.

2 "Active BCAS Detailed Collision Avoidance Algorithms", The MITRE Corporation, McLean, Virginia, MTR-80W286, October 1980.

- It will handle conflicts involving any combination of avionics equipage.

- It will handle coordination between adjacent ground sites with overlapping coverage.

- It will coordinate between two aircraft each having the airborne collision avoidance system, when both are outside ground radar coverage.

- It will ensure continuity of instructions when one or both aircraft transition into or out of radar coverage, resulting in a transfer of responsibility from the airborne system to a ground system or vice versa.

The basic principle of the coordination logic is this. The system responsible for resolving a new two-aircraft conflict will first learn of all instructions currently being displayed to the pilot in each aircraft. It will then pick instructions for the new conflict consistent with the constraints represented by the pre-existing instructions. Air-to-air and air-to-ground data exchanges are used to carry out this process. The process is also supported by appropriate protocols and read/write semaphores.

The coordination logic that resulted was rather complex. The coordination logic contained within the airborne collision avoidance system consisted of approximately 40 pages of detailed flow charts. The logic for the simple avionics device that receives instructions from the ground consisted of 5 pages of detailed flow charts. Verifying the coordination logic was difficult because of the many combinations of variables. Each aircraft could have one of three types of avionics equipage and each aircraft could initially be outside of radar coverage or in the radar coverage of one or more radars. There were many events that could happen during the course of a conflict. Each airborne collision avoidance system and ground system could detect the beginning or the end of a conflict. An aircraft could transition into or out of radar coverage. An airborne system or a ground system could fail to establish communications on any given attempt. Other events were also possible. The number of different sequences with which all of these events could possibly occur was extremely large.

The FAA desired to conduct a comprehensive verification of the coordination logic for the following reasons.

- The overall logic is rather complex. It is a distributed logic which deals with real-time processing in an interrupt environment.

- A fault in the coordination logic could preclude the collision avoidance systems from preventing a midair collision. Other types of faults could possibly generate erroneous instructions that could cause a hazardous situation when one would not have existed without the collision avoidance systems. Thus, the safety of the software design was of concern.

- Correcting a software fault after implementation would be very difficult. The avionics would potentially exist in thousands of aircraft.

- The aircraft owners and operators were skeptical about the safety of the coordination logic.

The FAA initiated several activities directed toward verification of the coordination logic:

- A discrete event simulation which exercised actual coding of the coordination logic. Some of the variables in this simulation were varied over all possible values. Others had values selected randomly. Approximately 1000 scenarios were tested. This is reported in Reference 3.

- Monte Carlo testing of the entire logic of the airborne collision avoidance system. The main interest in this simulation was in testing the resolution capability of the instructions generated. But in the process, the most common paths of the coordination logic were tested thousands of times.

- Limited flight tests of the coordination logic were conducted on the complete hardware when flown in two aircraft.

- An analytical safety study of the coordination logic was conducted using adaptations of some of the traditional system safety techniques. This study is the subject of the following section.

After these studies had been completed, the FAA changed the course of its aircraft collision avoidance program. A decision was made not to implement the ground collision avoidance system and some changes were made to the airborne collision avoidance system. The coordination logic currently used to ensure compatibility between two airborne collision avoidance systems has been carried essentially unchanged from the previous logic. It has benefited from the verification study reported below.

---

[3] "Discrete Event Simulation of the Resolution Advisory Register", The MITRE Corporation, McLean, Virginia, MTR-82W27, May 1982.

A Case Study

The authors were assigned the task of conducting a safety study of the coordination logic. From prior familiarity with system safety studies, it seemed that adaptation of some of the system safety techniques to this task could be productive. At the time the study was begun, the authors did not know whether techniques such as fault tree analysis or event tree analysis had previously been applied to the study of software. The approach used was refined as the study proceeded. The final approach is described below.

A single analyst with rather extensive applications programming experience was assigned to the task. The task took approximately eight months to complete. The analyst had no part in the design of the coordination logic and had no familiarity with it prior to beginning this project. Neither did he have prior experience conducting system safety analyses.

The analyst first acquired background in the traditional system safety techniques and consulted with several professionals who had experience in this field. He then spent approximately one month becoming thoroughly familiar with the coordination design. This design was presented at the detailed flow chart level and was accompanied by a reasonable amount of explanatory text.

The next step was to prepare a list of failure vulnerabilities. This step is comparable to determining the top level events in traditional fault tree analysis. The difference is that both top level events and initiating events were originally placed on this list. Items of special concern to members of the technical community were included. Additional items were solicited from designers and other testers of the logic. The collection of software trouble reports from the development of the coordination logic was consulted to learn what types of errors had been made previously in this area. This suggested several more items for the list. Other new items were added as a result of trains of thought established while conducting in-depth analyses for initial items on the list.

The initial process of generating the list was more or less one of brainstorming. Any hazy area of concern was listed. In the beginning, there was no interest in characterizing the exact nature of the failure vulnerability; the major goal was to collect a varied list of potential trouble areas. Then the definitions of items on the list of failure vulnerabilities were sharpened and the boundaries between different related areas were made clearer. The list was then partitioned into two lists -- one a list of undesired end results, the other a list of initiating events.

As is common in system safety studies, two types of analyses were conducted -- cause oriented analyses and consequence oriented

analyses. Figure 2 shows the relationship between the two and provides an example of each. In cause oriented analysis, each item on the list of undesired end results is used to begin the development of a logic tree. The development proceeds in what might be considered a backward direction, starting with the undesired end result and searching for a basic software fault that could cause that undesired end result. In a similar way, each item on the list of initiating events is used to begin a logic tree for consequence oriented analysis. The analysis proceeds in a forward direction looking at a series of consequences of the initiating event, all the while searching for a software fault that could lead to a hazardous situation.

Figure 3 shows the list of failure vulnerabilities analyzed in this study. It should be remembered that these were hypothesized faults, not actual faults found in the coordination logic. Most of these were eventually found not to be faults. No claim is made that this list is exhaustive. It was not intended that the study be exhaustive; it was a goal, however, to cover as many types of events as possible. In presenting these results to several audiences, it has been found that members of the audience have suggested additional items for the list. Most of these suggestions were found to have been included as lower level branches in a tree drawn for one of the items already on the list. Those that were new were added to the list. These suggestions were valuable for extending the coverage of the study.

A tree of either the cause oriented or the consequence oriented type was developed as deeply as possible. Then, a single path leading from the top of the tree down to a single terminating branch was taken as a single situation for study. All of the blocks in this path fully describe that specific situation. The analyst would then trace his way through the coordination logic using all of the required flow charts in the proper order as called for by this particular path in the logic tree. (The authors have called this process "finger tracing".) He would observe whether or not the logic worked correctly in that particular situation. If so, he would go on to the next path in the logic tree. If not, he had found a software fault and he would document this fault in detail.

The authors did not attempt to use probabilities in this analysis. The purpose of this analysis was to find as many unrecognized software errors as possible. When errors were found, they were corrected. The goal in the design and verification process was to achieve a state where there were no known software errors outstanding. Probabilistic analysis was not relevant to the objectives of this study. The authors feel that the use of probabilities would not be successful in any software verification effort of this type. This type of analysis has nothing to contribute toward estimating probabilities of encountering software

errors or estimating the number of remaining software errors in a given program.

To assist in these analyses, a checklist was tailored specifically for this project. Checklists from a number of other sources (particularly Reference 4) were consulted in creating this tailored one. The checklist was also enhanced and refined as the study proceeded. The checklist was used in two ways. It was used to suggest additional causes or consequences that would feed into a particular block in a tree. That is, when one is looking at a particular block and is determining whether the tree can be branched out to one level deeper, the checklist can suggest new causes leading to that block. The second use of the checklist is to help the analyst think of situations which might lead to a fault as he is "finger tracing" the logic through the flow charts. When he has completed a flow chart, he can quickly review the checklist to see if any of the items mentioned there would create a fault in that flow chart with the specific situation just analyzed. The checklist used in this study is shown in Figure 4.

An example of a cause oriented logic tree is presented in Figure 5. This is part of the tree from the undesired end event "Two Instructions With Incompatible Senses Displayed to Pilot at the Same Time". The left hand path through this tree represents a major fault that was discovered through use of consequence oriented analysis. The situation represented by this path is that a "Climb" and a "Descend" have simultaneously been displayed to the pilot of an aircraft carrying the airborne collision avoidance system. The aircraft is simultaneously in conflict with two other aircraft and the airborne system has generated a "Climb" against one threat and a "Descend" against the other. Normally, the logic tests any potential new instruction for compatibility with existing instructions before displaying it. If an incompatibility exists it will modify both the new and the existing instruction to yield an allowable instruction such as "Don't Climb and Don't Descend" (i.e., maintain level flight). In the logic tree it was hypothesized that there was an error in the compatibility subroutine. Using the conditions as described in this situation, the analyst traced the logic flow through the appropriate flow charts. He used the checklist to help search for a software fault that could generate the conditions described in the logic tree. He found such a fault, which had not been discovered in any of the other types of testing conducted on the coordination logic.

The reader will observe that the logic tree of Figure 5 is similar to a fault tree. The top is an undesired end event and the bottom is a cause leading to that event. The logic tree uses "and" and "or" gates. The tree differs from a fault tree, however, in that each box in the logic tree is not a fault. In the logic tree, some

[4] The Art of Software Testing, Myers, Glenford, J., Wiley, New York, 1979.

of the boxes describe various attributes of the encounter. In a normal fault tree, when all of the boxes on a particular path are listed together from top to bottom, there is generally a clear order or sequence of faults. In the logic tree, successive levels may simply describe different attributes of the encounter, for which there is no real or implied order. In the logic tree, all boxes at the same level exhaust all possible states of the attribute being described. The authors freely admit that they have combined some simple exhaustive enumeration techniques with the traditional fault tree approach, in the process sacrificing some purity in the fault tree form. But this approach seems convenient for software.

Figure 6 presents an example of a consequence oriented logic tree. It represents a portion of the tree constructed for the initiating event "A Threat's Airborne System Fails Abruptly During a Conflict". In preparing Figure 6, a great deal of exhaustive enumeration was carried out on the side. This analysis showed that, as far as the coordination logic was concerned, all possible relationships between the two events, "Own Detects the Beginning of the Conflict With the Threat" and "The Threat's Airborne Collision Avoidance System Fails" could be represented by six statements, "Own Detects Conflict Before Failure", "Own Detects on Same Cycle as Failure", etc. (The airborne system goes through all of its calculations once per second. A single pass through all of the calculations is called a cycle.) For each of these six conditions, it was possible to observe the logic flow for one cycle of own's system at any time with respect to the time when the threat's system failed. It was found that all of these possibilities collapsed to just five, which are also shown in Figure 6.

It was then possible to trace the flow of the logic in the software flow charts for each of the paths leading to a low level branch in the logic tree. The analyst would annotate each lower level branch with the results of that study. The notes indicate the pages in the flow chart document and the names for all subroutines traced and indicate the results. For the path leading to the block reading "Process Observed in SELADV", no fault was found. The note for the other low level branch shows that a minor fault was found in the subroutine, SENDINT. The surveillance subsystem of the airborne system dropped the track for the threat several seconds earlier than the collision avoidance subsystem. As a result, it was possible for the collision avoidance subsystem to ask the surveillance subsystem to try to send a message to the threat, when the surveillance subsystem no longer had any record of that threat. The results in the software would be undefined.

In some cases the analyst would come to a potential failure situation which he would judge to be so improbable that he would elect not to study it further. He would also write a note to this

effect adjacent to that block. In other cases, there were limitations which were known to exist in the design. They had been accepted as tolerable; hence, they were not treated as faults.

Another type of notation was made at a low level branch when a particular feature of the coordination logic represented a trap or protection against the particular hypothesized fault being studied. For example, suppose the analyst has been considering the possibility that a data structure might become locked out indefinitely. He comes to a place in a flow chart where a timer unlocks the data structure after a time-out period. If he has verified that the timer works correctly, he can make a note at the appropriate block in the logic tree indicating that the fault is protected by a particular time-out mechanism.

Results

Over 90 pages of detailed logic trees were constructed during this study. As a result of this effort, 10 minor software errors were uncovered in the coordination documentation. These were generally cases where what the designer intended was correct; there was simply a documentation error. In addition, there was one significant design flaw identified. Finding this error was directly attributable to use of cause oriented analysis. All of these errors were corrected, and selected parts of the verification were repeated with the result that no new errors were found.

The authors recognize that this was not an exhaustive study. They do not claim that there are no other hidden software faults. However, the use of cause oriented and consequence oriented analysis in this study was satisfying. It gave the authors the feeling that the coordination logic had been very thoroughly studied from many different perspectives. And it was felt that the approach used provided a helpful structure that was more effective than an unfocused manual checking of the logic.

The authors found that the approach used in this study had the following positive attributes for verifying software.

- Cause oriented analysis is a powerful tool for identifying previously unrecognized software faults. The basic concept of hypothesizing an undesired end event and then of developing a tree, by trying to imagine all of the ways in which that event could come about, is quite effective. This is because it represents a completely different point of departure and provides a different perspective from most traditional methods of verifying software. It helps the verifier identify new software faults. For this reason, it represents a very useful adjunct to other software

verification approaches. The deductive thought process represented by cause oriented analysis is undoubtedly the most significant contribution which system safety techniques can make toward verification of software used in safety-critical applications.

- Consequence oriented analysis is a further enhancement to an overall software verification effort, because it approaches the task from yet another point of view. For this reason, it is likely to find other software errors not found by cause oriented analysis.

- Use of a checklist for studies of this type is extremely helpful. Reviewing other checklists or other project software error histories in creating a tailored checklist is a way of capitalizing on previous experience, and of avoiding those errors that are made over and over again.

- The list of failure vulnerabilities provides a convenient framework for studies of this type. The brainstorming approach makes it easy to get started. The first items on the list provide concrete situations for beginning the logic trees. Having critics or other persons suggest items for this list is also useful. Being able to add new items that are triggered during the analysis of the initial items on the list is quite helpful in extending the study. After the study, the list is useful as a way of succinctly conveying to an audience the scope of the study.

- The logic tree techniques show graphically the benefit of any built-in failure protection mechanisms in the design. They could also suggest to the software designer where additional "brickwalling" would be useful.

- The resulting tree diagrams provide a clear record of design limitations which are known and accepted.

- These techniques are very useful for finding faults involving hardware and software interaction. They can be especially useful for studying how the software responds to various hardware failure modes.

The authors observed the following limitations of the system safety techniques as used in this study.

- These techniques provide no guarantee that all errors will be found.

- It is difficult to organize the logic trees in a satisfying way. Several iterations are required to word the blocks correctly and to organize the levels logically.

- Preparation of the logic trees is not mechanical. It requires a large measure of human judgment, imagination and skill. To be effective, the process takes a long time.

- The logic trees do not easily handle complex sequences of events. It was found, however, that these could sometimes be handled successfully on the side through exhaustive enumeration using tables or matrices. A reduced number of situations could then be represented in the logic trees.

- These techniques are not useful for quantitative analysis or for the assessment of probabilities. They offer no solution to the problem of how to estimate the overall probability of a critical failure for an entire system which has physical components as well as software.

Recommendations for Verifying Other Software Systems

The authors feel that a good program for evaluating safety-critical software should use a variety of verification techniques. Each technique should be able to uncover software faults not found by the others. The budget for verifying the software should be partitioned so as to support, where feasible, design walk-throughs, unit tests, simulation, real-time bench tests with an environment simulator, live or flight tests, and a system safety analysis program. The system safety program should include the following.

- A list of potential vulnerabilities.

- Cause oriented analysis.

- Consequence oriented analysis.

- A well developed and tailored checklist.

- Frequent use of exhaustive enumeration techniques wherever feasible to generate all possible combinations of variables or conditions. It is generally not possible to do this for the software system as a whole, but it can be done for local conditions at specific places in the logic trees.

- A feedback mechanism to enlarge the list of potential vulnerabilities as the analysis proceeds.

The authors recommend that the scope of the system safety analysis be pared down so that the study concentrates on only the

safety critical areas of the entire system. A good system design for a large system would ensure that only a portion of the total software would be safety critical and that this portion could be largely isolated from the remainder of the software. The safety analysis should first identify those critical areas that require concentrated attention.

It is further recommended that the system safety analysis be conducted relatively early in the software development cycle. It could probably be successfully accomplished at the B-specification or the C-specification level using either flow charts or program design language descriptions. In any case, it should probably be done prior to the beginning of coding because any errors found are easier to correct at this point. The authors feel it would be rather difficult to conduct such a study using actual source code, because the total number of pages involved would make "finger tracing" difficult.

References

1. "Report of the FAA Task Force on Aircraft Separation Assurance", Federal Aviation Administration, Report No. FAA-EM-78-19, Vols. I, II, III, January 1979.

2. "Active BCAS Detailed Collision Avoidance Algorithms", The MITRE Corporation, McLean, Virginia, MTR-80W286, October 1980.

3. "Discrete Event Simulation of the Resolution Advisory Register", The MITRE Corporation, McLean, Virginia, MTR-82W27, May 1982.

4. The Art of Software Testing, Myers, Glenford, J., Wiley, New York, 1979.

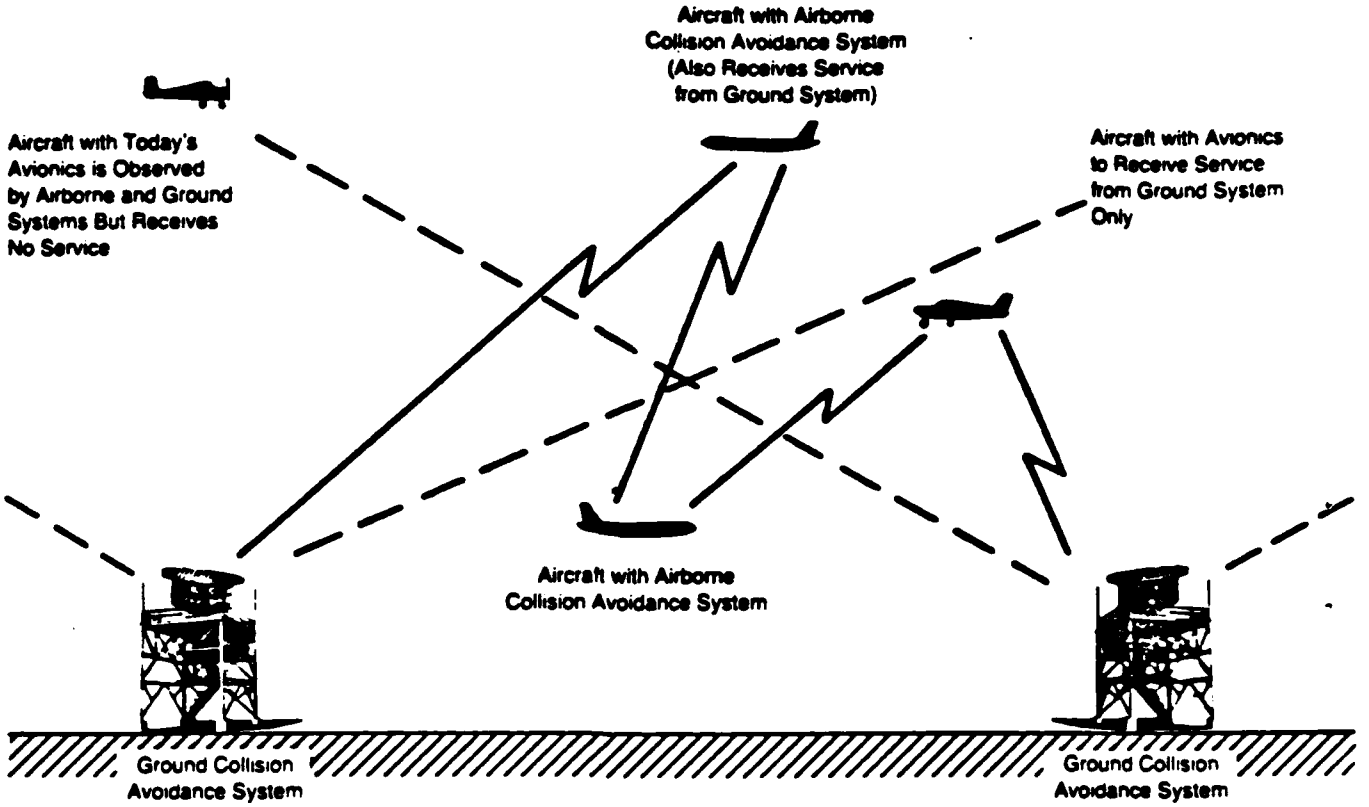


FIGURE 1
ELEMENTS OF THE AIRCRAFT COLLISION AVOIDANCE COORDINATION LOGIC

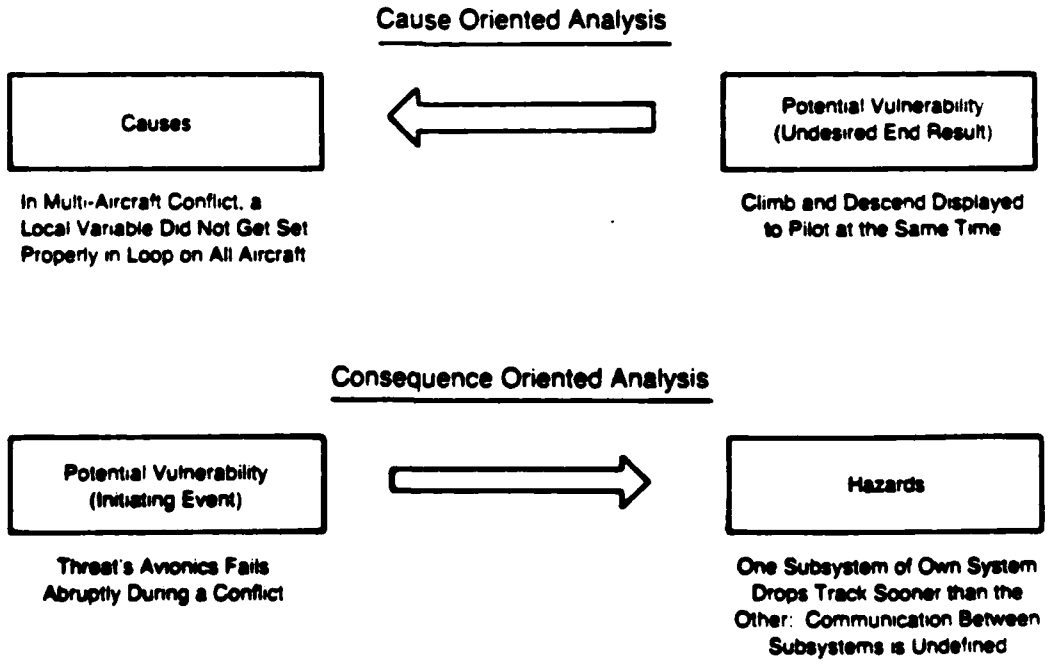


FIGURE 2
CONTRAST BETWEEN CONCEPTS OF CAUSE ORIENTED AND CONSEQUENCE ORIENTED ANALYSIS

- Undesired End Results Addressed by Cause Oriented Analysis
  - No Instruction Displayed to Own Pilot When Warranted Due to Coordination Failure
  - Another Airborne System Runs Wild and Causes an Invalid Instruction to be Displayed to Own Pilot
  - An Instruction Displayed to Own Pilot is Not Coordinated With Instruction Displayed to Other Pilot
  - No System (Airborne/Ground) Accepts Responsibility for a Given Conflict
  - Two Systems Take Responsibility for a Given Conflict at the Same Time
  - An Instruction Remains Displayed Indefinitely
  - The Data Structure Remains Locked Indefinitely
  - An Indefinite Wait Occurs While Waiting for Coordination to Complete
  - Two Instructions With Incompatible Senses Displayed to Pilot at the Same Time
  - One System Requests the Data Structure at Nearly the Same Time as Another and Mistakenly Thinks the Reply was Intended for It
  - Circular Deadlock Effect With Three Aircraft Simultaneously Interrogating Each Other

- Initiating Events Addressed by Consequence Oriented Analysis
  - A Threat's Airborne System Fails Abruptly During a Conflict
  - Ground System Computer Fails Abruptly
  - Own Airborne System Transmitter/Receiver Fails
  - Threat's Airborne System Transmitter/Receiver Fails
  - Position Correlation Algorithm Makes Correlation With Wrong Track
  - Position Correlation Algorithm Fails to Make Correlation With Proper Track

FIGURE 3
THE FAILURE VULNERABILITIES ANALYZED IN THIS STUDY

- Undesirable Effects from Incoming Message Interrupt?
- Undesirable Effects of Restarting in the Middle of a Subroutine After Processing an Interrupt?
- Undesirable Effects if a Process Does Not Complete in Available Time?
- Was a Critical Variable Properly Initialized?
- Possibility for an Infinite Loop?
- Are Linked Data Structures Cleaned Up When a Main Data Structure is Deleted?
- Undesirable Effects if Track Broken and Restarted During a Conflict?
- Undesirable Effects if Intruder's Avionics Equipage Changes During a Conflict?
- Pilot Manual Control Idiot-Proof?
  - Own Pilot
  - Other Pilot
- Undesirable Effects if Transition Into or Out of Ground Radar Coverage Occurs During a Conflict?
- Are Multi-Aircraft Effects Considered?
- Are All Required Internal Data Structures Updated When an Update is Required?
- Is the External Data Structure Updated Appropriately?

FIGURE 4
CHECKLIST USED IN THIS STUDY

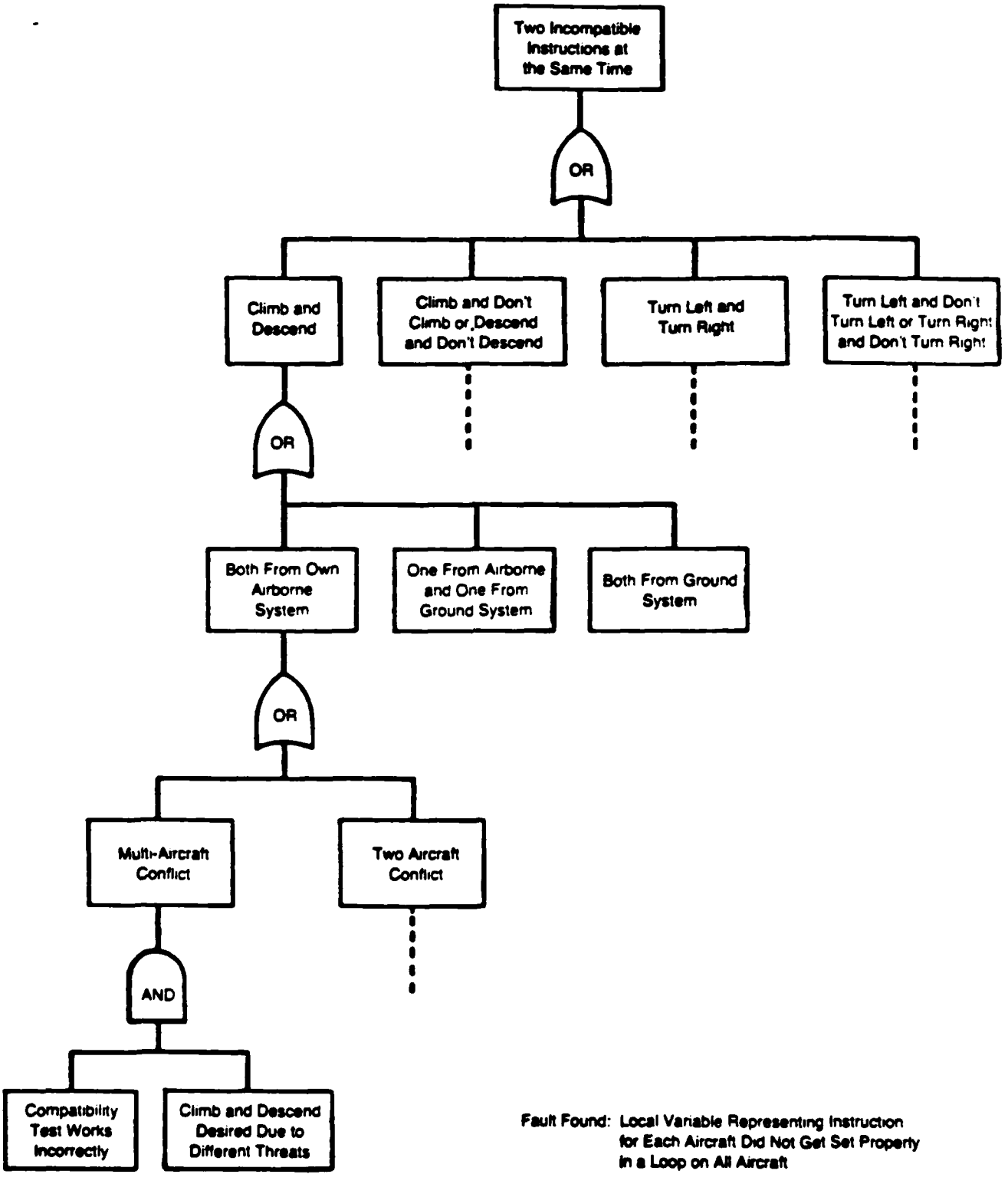


FIGURE 5
EXAMPLE LOGIC TREE FOR CAUSE ORIENTED ANALYSIS

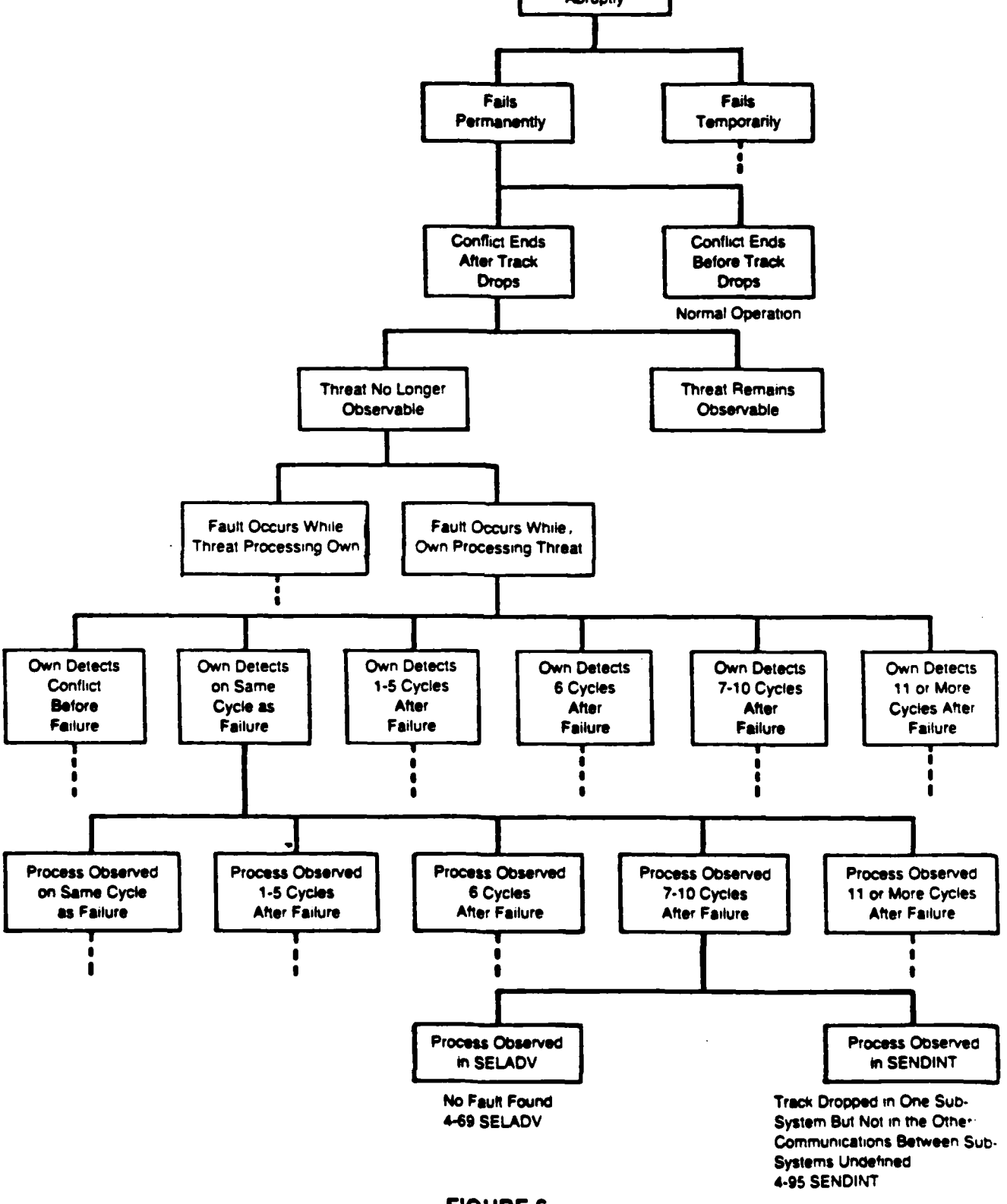


FIGURE 6
EXAMPLE LOGIC TREE FOR CONSEQUENCE ORIENTED ANALYSIS

END

DTIC

9-86